

4

EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

In this chapter we instantiate the usage of Software Diversification for offensive and defensive purposes. We present two selected use cases that exploit Software Diversification through our technical contributions presented in Chapter 3. The selected cases are representative of applications of Software Diversification for WebAssembly in browsers and standalone engines.

4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [?]. This is due to the essence of cryptojacking, the faster the mining, the better. Although the research of Lehmann and colleagues [?] suggests a decline in browser-based cryptominers, mainly due to the shutdown of Coinhive, a 2022 report by Kaspersky indicates that the use of cryptominers is on the rise [?]. This underscores the ongoing need for effective automatic detection of cryptojacking malware.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [?]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Subsection 2.1.6. However, it's worth noting that malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [?], highlighting the critical importance of this use case. For an in-depth discussion on this topic, we direct the reader to our

⁰Comp. time 2023/10/08 12:05:05

contribution [?]. The use of case illustrated in the subsequent sections uses Offensive Software Diversification for the sake of evading malware detection in WebAssembly.

4.1.1 Threat model: cryptojacking defense evasion

Let us illustrate how a malicious Wasm binary could be involved into browser cryptojacking. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

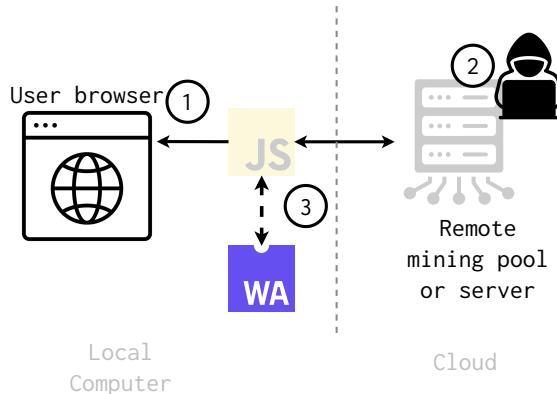


Figure 4.1: Cryptojacking components.

For the previous triad to work, the following steps are executed. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

Considering the previous scenario, several techniques, as outlined in Subsection 2.1.6, can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such defense scenario is illustrated in Figure 4.2, where the WebAssembly malicious binary is blocked in ③. The primary aim of our use of case is to investigate the effectiveness of code diversification as a means to circumvent cryptojacking defenses. Specifically, we

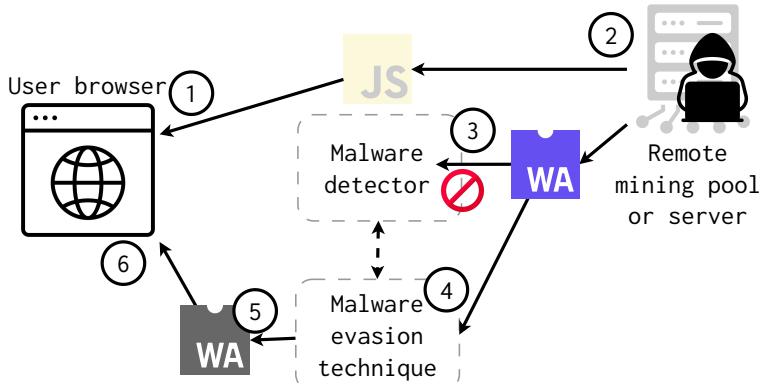


Figure 4.2: Taken from [?]

assess whether the following evasion workflow can successfully bypass existing security measures:

1. The user lands on a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to (1) and (2) in Figure 4.2. Notice that, various methods can be used to inject cryptojacking malware, including malicious browser extensions, malvertising, compromised websites, or deceptive links [?].
2. A malware detection mechanism(malware oracle) identifies and blocks malicious WebAssembly binaries at (3). For example, a network proxy could intercept and forward these resources to an external detection service via its API.
3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at (4).
4. The attacker delivers the modified binary instead of the original one (5), which initiates the cryptojacking process and compromises the browser (6). The detection method is completely oblivious to the malicious nature of the binary, and the attack is successful.

4.1.2 Methodology

In this study, we categorize malware detection mechanisms as malware oracles, which can be of two types: binary and numeric. A binary oracle provides a binary decision, labeling a WebAssembly binary as either malicious or benign. In

contrast, a numeric oracle returns a numerical value representing the confidence level of the detection.

Definition 5. *Malware oracle* A malware oracle is a detection mechanism that returns either a binary decision or a numerical value indicating the confidence level of the detection.

For empirical validation, we employ VirusTotal as a numeric oracle and MINOS [?] as a binary oracle. VirusTotal is an online service that analyzes files and returns a confidence score, thus qualifying as a numeric oracle. MINOS, on the other hand, converts WebAssembly binaries into grayscale images and employs a convolutional neural network for classification. It returns a binary decision, making it a binary oracle.

We use the wasmbench dataset [?] to establish a ground truth. After running the wasmbench dataset through VirusTotal and MINOS, we identify 33 binaries flagged as malicious by at least one VirusTotal vendor and also detected by MINOS.

To simulate the evasion scenario, we use WASM-MUTATE to generate WebAssembly binary variants to evade malware detection. We use WASM-MUTATE in two configurations: controlled and uncontrolled diversification.

Definition 6. *Uncontrolled Diversification* Unlike controlled diversification, uncontrolled diversification is a stochastic process where each transformation has an equal likelihood of being applied to the input WebAssembly binary.

Definition 7. *Controlled Diversification* In controlled diversification, the transformation process of a WebAssembly program is guided by a numeric oracle, which influences the probability of each transformation. For instance, WASM-MUTATE can be configured to apply transformations that minimize the oracle’s confidence score. Note that controlled diversification needs a numeric oracle.

Based on the two types of malware oracles and diversification configurations, we examine three scenarios: 1) VirusTotal with a controlled diversification, 2) VirusTotal with an uncontrolled diversification, and 3) MINOS with an uncontrolled diversification. Notice that, the fourth scenario with MINOS and a controlled diversification is not feasible, as MINOS is a binary oracle and cannot provide the numerical values required for controlled diversification.

Our evaluation focuses on two key metrics: the success rate of evading detection mechanisms in VirusTotal and MINOS across the 33 flagged binaries, and the performance impact on the variants that successfully evade detection. The first metric measures the efficacy of WASM-MUTATE in bypassing malware detection systems. For each flagged binary, we input it into WASM-MUTATE, configured with the selected oracle and diversification strategy. We then iteratively apply transformations to the output from the preceding step. This iterative process is halted either when the binary is no longer flagged by the

oracle or when a maximum of 1000 stacked transformations have been applied. This process is repeated with 10 random seeds per binary to simulate 10 different evasion experiments per binary. The second metric is crucial for validating the real-world applicability of WASM-MUTATE in evading malware detection. Specifically, if the evasion process significantly degrades the performance of the resulting binary compared to its original version, it becomes less likely to be employed in practical scenarios, such as cryptojacking. For this, we take the variants that fully evade VirusTotal when generated with WASM-MUTATE in controlled and uncontrolled diversification configurations.

4.1.3 Results

In Table 4.1, we present a comprehensive summary of the evasion experiments presented in [?], focusing on two oracles: VirusTotal and MINOS[?]. The table is organized into two main categories to separate the results for each malware oracle. For VirusTotal, we further subdivide the results based on the two diversification configurations we employ: uncontrolled and controlled diversification. In these subsections, we provide columns that indicate the number of initial detections (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which the uncontrolled diversification or controlled diversification setups are better than each other, the lower, the better. The MINOS section simply includes a column that specifies the number of transformations needed for complete evasion. The table has $33 + 1$ rows, each representing a unique Wasm malware study subject. The final row offers the median number of transformations required for evasion across all setups and oracles.

Uncontrolled diversification to evade VirusTotal: We run uncontrolled diversification with WASM-MUTATE with a limit of 1000 iterations per binary. At each iteration, we query VirusTotal to check if the new binary evades the detection. This process is repeated with 10 random seeds per binary to simulate 10 different evasion experiments per binary. As shown in the uncontrolled diversification part of Table 4.1, we successfully generate variants that evade detection for 30 out of 33 binaries. The mean value of iterations needed to generate a variant that evades all detectors ranges from 120 to 635 stacked transformations. The mean number of iterations needed is always less than 1000 stacked transformations. There are 3 binaries for which the uncontrolled diversification setup does not completely evade the detection. In these three cases, the algorithm misses 5 out 31, 6 out of 30 and 5 out 26 detectors. The explanation is the maximum number of iterations 1000 we use for our experiments. However, having more iterations seems not a realistic scenario. For example, if some transformations increment the binary size during the transformation, a considerably large binary might be impractical for bandwidth reasons. Overall,

| Hash | #D | VirusTotal | | | MINOS[?] | |
|-----------|----|------------------------------|-------------|----------------------------|-------------|-----|
| | | Uncontrolled diversification | | Controlled diversification | | |
| | | Max. evaded | Mean trans. | Max. evaded | Mean trans. | |
| 47d29959 | 31 | 26 | N/A | 19 | N/A | 100 |
| 9d30e7f0 | 30 | 24 | N/A | 17 | N/A | 419 |
| 8ebf4e44 | 26 | 21 | N/A | 13 | N/A | 92 |
| c11d82d | 20 | 20 | 355 | 20 | 446 | 115 |
| 0d996462 | 19 | 19 | 401 | 19 | 697 | 24 |
| a32a6f4b | 18 | 18 | 635 | 18 | 625 | 1 |
| fbdd1efa | 18 | 18 | 310 | 18 | 726 | 1 |
| d2141ff2 | 9 | 9 | 461 | 9 | 781 | 81 |
| aaffff587 | 6 | 6 | 484 | 6 | 331 | 1 |
| 046dc081 | 6 | 6 | 404 | 6 | 159 | 33 |
| 643116ff | 6 | 6 | 144 | 6 | 436 | 47 |
| 15b86a25 | 4 | 4 | 253 | 4 | 131 | 1 |
| 006b2fb6 | 4 | 4 | 282 | 4 | 380 | 1 |
| 942be4f7 | 4 | 4 | 200 | 4 | 200 | 29 |
| 7c36f462 | 4 | 4 | 236 | 4 | 221 | 85 |
| fb15929f | 4 | 4 | 297 | 4 | 475 | 1 |
| 24aae13a | 4 | 4 | 252 | 4 | 401 | 980 |
| 000415b2 | 3 | 3 | 302 | 3 | 34 | 960 |
| 4cbdbbb1 | 3 | 3 | 295 | 3 | 72 | 1 |
| 65debcbe | 2 | 2 | 131 | 2 | 33 | 38 |
| 59955b4c | 2 | 2 | 130 | 2 | 33 | 38 |
| 89a3645c | 2 | 2 | 431 | 2 | 107 | 108 |
| a74a7cb8 | 2 | 2 | 124 | 2 | 33 | 38 |
| 119c53eb | 2 | 2 | 104 | 2 | 18 | 1 |
| 089dd312 | 2 | 2 | 153 | 2 | 123 | 68 |
| c1be4071 | 2 | 2 | 130 | 2 | 33 | 38 |
| dceaf65b | 2 | 2 | 140 | 2 | 132 | 66 |
| 6b8c7899 | 2 | 2 | 143 | 2 | 33 | 38 |
| a27b45ef | 2 | 2 | 145 | 2 | 33 | 33 |
| 68ca7c0e | 2 | 2 | 137 | 2 | 33 | 38 |
| f0b24409 | 2 | 2 | 127 | 2 | 11 | 33 |
| 5bc53343 | 2 | 2 | 118 | 2 | 33 | 33 |
| e09c32c5 | 1 | 1 | 120 | 1 | 488 | 15 |
| Median | | | 218 | | 131 | 38 |

Table 4.1: TODO description and main takeaway

uncontrolled diversification with WASM-MUTATE clearly decreases the detection rate by VirusTotal antivirus vendors for cryptojacking malware, achieving total evasion of WebAssembly cryptojacking malware in 30/33 (90%) of the malware dataset. When compared with the data in Table 3.1, it becomes evident that WASM-MUTATE generates an average of nearly 10000 variants per binary within an hour. Thus, WASM-MUTATE is capable of successfully evading detection systems in just a matter of minutes.

Controlled diversification to evade VirusTotal: Uncontrolled diversification does not guide the diversification based on the number of evaded detectors, it is purely random, and has some drawbacks. For example, some transformations might suppress other transformations previously applied. We have observed that, by carefully selecting the order and type of transformations applied, it is possible to evade detection systems in fewer iterations. This can be appreciated in the results of the controlled diversification part of Table 4.1. Analyzing the data in Table 4.1, we observe that the controlled diversification setup successfully generates variants that totally evade the detection for 30 out of 33 binaries, it thus as good as the uncontrolled setup. The iterations needed for the controlled diversification setup are 92% of the needed on average for the uncontrolled diversification setup. For 21 of 30 binaries that evade detection entirely, we observe that the mean number of oracle calls needed is lower than those in the baseline evasion algorithm. For example, f0b24409 needs 11 oracle calls with controlled diversification setup to fully evade VirusTotal, while for the uncontrolled one, it needs 127 oracles calls. For those 21 binaries, it needs only 40% of the calls the controlled diversification setup needs.

Uncontrolled diversification to evade MINOS: Additionally, relying solely on VirusTotal as the detection mechanism could be problematic, especially when specialized solutions exist exclusively for WebAssembly, unlike the general-purpose vendors in VirusTotal. To address this, we also assessed the efficacy of our evasion algorithms against MINOS, a WebAssembly-specific antivirus. We iteratively applied random mutations to the original malware binary until either MINOS was fully evaded or a maximum iteration limit was reached. This process was repeated 10 times for each binary. The outcomes are displayed in the last column of Table 4.1. The last row of Table 4.1 shows that WebAssembly diversification requires fewer iterations to evade MINOS than VirusTotal, meaning that it is easier to evade MINOS. The median number of iterations needed overall for evading VirusTotal is 218 for the uncontrolled diversification setup, and 131 for the controlled diversification setup, while for MINOS is 38. Remarkably, WASM-MUTATE totally evades detection for 8 out of 33 binaries in one single iteration in the case of MINOS. This shows that the MINOS model is fragile wrt binary diversification. According to those results, VirusTotal can be considered better than MINOS wrt to cryptojacking detection. The main reason is that a wider spectrum of antivirus vendors is used

in VirusTotal, while MINOS is a single detector.

Performance To evaluate the real-world efficacy of WASM-MUTATE in evading malware detection, we focused on six binaries that we could build and execute end-to-end, as these had all three components outlined in Figure 4.1. For these binaries, we replace the original WebAssembly code with variants generated using VirusTotal as the malware oracle and WASM-MUTATE for both controlled and uncontrolled diversification configurations. We then execute both the original and the generated variants. Given that the primary objective of cryptojacking is high-speed hash generation, we assessed the variants based on two key metrics: hash validity and hash generation rate.

We have found that 19% of the generated variants outperformed the original cryptojacking binaries. This improvement is attributed to WASM-MUTATE’s ability to introduce code optimizations (as discussed in Chapter 3). Additionally, debloating transformations, which eliminate unnecessary structures and dead code, resulted in a higher hash generation rate during the initial seconds of mining, likely due to faster compilation times. This suggests that focused optimization serves as a valuable tool for evasion in browsers.

On the contrary, 80% of the generated variants are less efficient than the original binary, with the least efficient variant operating at only 20% of the original hash generation rate. This performance drop is primarily due to non-optimal transformations introduced by WASM-MUTATE. Variants generated through uncontrolled diversification are generally slower. In summary, controlled diversification yielded variants that evaded VirusTotal detection with minimal performance overhead—the worst-performing variant was only 1.93 times slower than the original.

Contribution paper

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of our work are actionable, as we also provide quantitative evidence on specific malware transformations on which detection methods can focus. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" at *Computers & Security*, 2023 <https://www.sciencedirect.com/science/article/pii/S0167404823002067>.

4.2 Defensive Diversification: Speculative Side-channel protection

As previously discussed in Subsection 2.1.5, WebAssembly is rapidly gaining traction in backend environments. Companies like Cloudflare and Fastly are encouraging the use of WebAssembly in their edge computing platforms, allowing developers to deploy faster applications in a modular and securely sandboxed fashion. Typically, these client WebAssembly applications are designed as isolated services with a single, focused responsibility. This model is usually called Function-as-a-Service (FaaS) [? ?]. In Figure 4.3 we illustrate how WebAssembly binaries are used in FaaS platforms. Developers can submit any WebAssembly binary to the FaaS platform that are then replicated and moved around the Edge-cloud worldwide platform.

TODO Align the figure with the terms

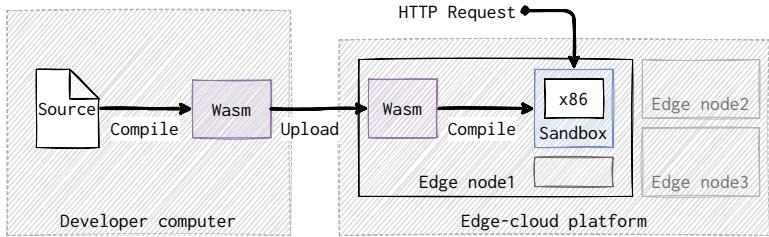


Figure 4.3: WebAssembly binaries on FaaS platforms.

The core concept behind Wasm in FaaS platforms is the ability to host thousands of client WebAssembly binaries within a single host process, which is then distributed across multiple servers and data centers. To achieve this, the platforms compile the Wasm programs to native code, which is then executed in a sandboxed environment. Then, the host processes are capable of instantiating a new Wasm sandbox for each client function, executing it in response to individual user requests in a matter of nanoseconds. Utilizing WebAssembly enables these platforms to inherently isolate the execution of client functions from one another as well as from the host process. However, this isolation is not foolproof against Spectre attacks [? ?].

In the subsequent sections, we illustrate how diversification can be used to protect WebAssembly binaries against Spectre attacks. We show a case of Defensive Software Diversification for the sake of protecting WebAssembly binaries.

4.2.1 Threat model: speculative side-channel attacks

Let us illustrate the threat model in which a WebAssembly program could be susceptible in FaaS platforms. Developers can submit any WebAssembly binary to the FaaS platform. This includes potential malicious actors that could upload a Wasm binary that, when compiled to native code, uses Spectre attacks to leak sensitive information from the host process. Spectre attacks exploit hardware predictors to induce misspredictions and speculatively execute instructions—gadgets that would not run sequentially. The attacker could then use this information to infer the contents of the memory of other client functions, or even the host process itself.

Narayan and colleagues [?] dissected the possible Spectre attacks for Wasm binaries into three categories based on the specific hardware predictor that is exploited and the specific FaaS scenario: Sandbox breakout attacks, Sandbox poisoning attacks and Host poisoning attacks. The first one, exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target. The second one takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition. The third and last one exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions. Each attack methodology relies on the extraction of memory bytes from another hosted Wasm binary that executes in the same host process.

TODO Add diagram **TODO** Explain here the threat model with the diagram

4.2.2 Approach

- Use of `wasm-mutate`

4.2.3 Results

- Diminshing of BER - Rockiki paper on portable side channel in browsers.
TODO TBD discuss deoptimization

4.2.4 Partial input/output validation

When WASM-MUTATE generates a variant, it can be executed to check the input/output equivalence. If the variant has a `_start` function, both binaries, the original and the variant can be initialized. If the state of the memory, the globals and the stack is the same after executing the `_start` function, they are partially equivalent.

The `_start` function is easier to execute given its signature. It does not receive parameters. Therefore, it can be executed directly. Yet, since a WebAssembly program might contain more than one function that could be indistinctly called with an arbitrary number of parameters, we are not able to validate the whole program. Thus, we call the checking of the initialization of a Wasm variant, a partial validation.

4.2.5 Some other works to be cited along with the paper.

Mostly in the Intro

Spectre and side-channel defenses

- paper 2021: Read this, since it is super related, https://www.iscure-journal.com/article_136367_a3948a522c7c59c65b65fa87571fde7b.pdf [?]

- A dataset of Wasm programs: [?]

- Papers 2020

- Papers 2019 - [?]

Selwasm: A code protection mechanism for webassembly Babble

- <https://arxiv.org/pdf/2212.04596.pdf>

Principled Composition of Function Variants for Dynamic Software Diversity and Program Protection

- <https://dl.acm.org/doi/10.1145/3551349.3559553>

How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes

- <https://ieeexplore.ieee.org/document/9975423>

Code obfuscation against symbolic execution attacks

Code artificiality: A metric for the code stealth based on an n-gram model

Semantics-aware obfuscation scheme prediction for binary

Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly

Synthesizing Instruction Selection Rewrite Rules from RTL using SMT "We also synthesize integer rewrite rules from WebAssembly to RISC-V "

Wafl: Binary-only webassembly fuzzing with fast snapshots

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* <https://arxiv.org/pdf/2309.07638.pdf>.

4.3 Conclusions

